下载APP

答疑1 | 第1~6讲课后思考题答案及常见问题解析

2021-10-23 蒋德钧

《Redis源码剖析与实战》 课程介绍 >

讲述：蒋德钧
时长 14:14 大小 13.05M

你好，我是蒋德钧。

咱们的课程已经快接近尾声了，之前我主要把精力和时间集中在了课程内容的准备上，没有来得及及时给大家做答疑，以及回复同学们提出的问题，在这也和同学们说一声抱歉，接下来我会尽快来回复大家的疑问。但其实，在这期间我看到了很多同学的留言，既有针对咱们课程课后思考题的精彩解答，也有围绕课程内容本身提出的关键问题，而且这些问题的质量很高，非常值得好好讨论一下。

那么，今天这节课，我就先来对课程的前 6 节的思考题做一次答疑。你也可以借此机会来回顾下咱们课程一开始时学习的内容，温故而知新。

第 1 讲

问题：Redis 从 4.0 版本开始，能够支持后台异步执行任务，比如异步删除数据，那么你能在 Redis 功能源码中，找到实现后台任务的代码文件吗？

关于这个问题，@悟空聊架构、@小五、@Kaito 等不少同学都给出了正确答案。我在这些同学回答的基础上，稍微做了些完善，你可以参考下。

Redis 支持三类后台任务，它们本身是在bio.h文件中定义的，如下所示：

复制代码

```
#define BIO_CLOSE_FILE    0    //后台线程关闭文件
#define BIO_AOF_FSYNC     1    //后台线程刷盘
#define BIO_LAZY_FREE     2    //后台线程释放内存
```

那么，在 Redis server 启动时，入口 main 函数会调用 InitServerLast 函数，而 InitServerLast 函数会调用 bioInit 函数，来创建这三类后台线程。这里的 bioInit 函数，则是在bio.c文件中实现的。

而对于这三类后台任务的执行来说，它们是在 bioProcessBackgroundJobs 函数（在 bio.c 文件中）中实现的。其中，BIO_CLOSE_FILE 和 BIO_AOF_FSYNC 这两类后台任务的实现代码，分别对应了 close 函数和 redis_fysnc 函数。而 BIO_LAZY_FREE 任务根据参数不同，对应了 lazyfreeFreeObjectFromBioThread、lazyfreeFreeDatabaseFromBioThread 和 lazyfreeFreeSlotsMapFromBioThread 三处实现代码。而这些代码都是在lazyfree.c文件中实现。

此外，还有一些同学给出了异步删除数据的执行流程和涉及函数，比如，@曾轼麟同学以 unlink 为例，列出了删除操作涉及的两个执行流程，我在这里也分享下。

unlink 实际代码的执行流程如下所示：

> 使用异步删除时的流程：unlinkCommand -> delGenericCommand -> dbAsyncDelete -> dictUnlink -> bioCreateBackgroundJob 创建异步删除任务 -> 后台异步删除。
>
> 不使用异步删除时的流程：unlinkCommand -> delGenericCommand -> dbAsyncDelete -> dictUnlink -> dictFreeUnlinkedEntry 直接释放内存。

此外，@悟空聊架构同学还提到了在 Redis 6.0 中增加的 IO 多线程。不过，Redis 6.0 中的多 IO 线程，主要是为了利用多核并行读取数据、解析命令，以及写回数据，这些线程其实是和主线程一起工作的，所以我通常还是把它们看作前台的工作线程。

第 2 讲

问题：SDS 字符串在 Redis 内部模块实现中也被广泛使用，你能在 Redis server 和客户端的实现中，找到使用 SDS 字符串的地方吗？

我们可以直接在全局变量 server 对应的结构体 redisServer 中，查找使用 sds 进行定义的成员变量，其代码如下所示：

复制代码

```
struct redisServer {
…
sds aof_buf;
sds aof_child_diff;
…}
```

同样，我们可以在客户端对应的结构体 client 中查找 sds 定义的成员变量，如下代码所示：

复制代码

```
typedef struct client {
…
sds querybuf;
sds pending_querybuf;
sds replpreamble;
sds peerid;
…} client;
```

此外，你也要注意的是，在 Redis 中，**键值对的 key 也都是 SDS 字符串**。在执行将键值对插入到全局哈希表的函数 dbAdd（在 db.c 文件）中的时候，键值对的 key 会先被创建为 SDS 字符串，然后再保存到全局哈希表中，你可以看看下面的代码。

复制代码

```
void dbAdd(redisDb *db, robj *key, robj *val) {
    sds copy = sdsdup(key->ptr);  //根据redisObject结构中的指针获得实际的key，调用s
```

```
3    int retval = dictAdd(db->dict, copy, val); //将键值对插入哈希表
4    …
5  }
```

第 3 讲

问题：Hash 函数会影响 Hash 表的查询效率及哈希冲突情况，那么，你能从 Redis 的源码中，找到 Hash 表使用的是哪一种 Hash 函数吗？

关于这个问题，@Kaito、@陌、@可怜大灰狼、@曾轼麟等不少同学都找到了 Hash 函数的实现，在这里我也总结下。

其实，我们在查看哈希表的查询函数 dictFind 时，可以看到它会调用 dictHashKey 函数，来计算键值对 key 的哈希值，如下所示：

复制代码

```
1 dictEntry *dictFind(dict *d, const void *key) {
2     ...
3     // 计算 key 的哈希值
4     h = dictHashKey(d, key);
5     ...
6 }
```

那么，我们进一步看 dictHashKey 函数，可以发现它是在 dict.h 文件中定义的，如下所示：

复制代码

```
1 #define dictHashKey(d, key) (d)->type->hashFunction(key)
```

从代码可以看到，dictHashKey 函数会实际执行哈希表类型相关的 hashFunction，来计算 key 的哈希值。所以，这实际上就和哈希表结构体中的 type 有关了。

这里，我们来看看哈希表对应的数据结构 dict 的定义，如下所示：

复制代码

```
1 typedef struct dict {
```

```
    dictType *type;
    ...
} dict;
```

dict 结构体中的成员变量 type 类型是 dictType 结构体，而 dictType 里面，包含了哈希函数的函数指针 hashFunction。

复制代码

```
typedef struct dictType {
    uint64_t (*hashFunction)(const void *key);
    ...
} dictType;
```

那么，既然 dictType 里面有哈希函数的指针，所以比较直接的方法，就是去看哈希表在初始化时，是否设置了 dictType 中的哈希函数。

在 Redis server 初始化函数 initServer 中，会对数据库中的主要结构进行初始化，这其中就包括了对全局哈希表的初始化，如下所示：

复制代码

```
void initServer(void) {
...
for (j = 0; j < server.dbnum; j++) {
        server.db[j].dict = dictCreate(&dbDictType,NULL);  //初始化全局哈希表
        ...}
}
```

从这里，你就可以看到**全局哈希表对应的哈希表类型是 dbDictType**，而 dbDictType 是在server.c文件中定义的，它设置的哈希函数是 dictSdsHash（在 server.c 文件中），如下所示：

复制代码

```
dictType dbDictType = {
    dictSdsHash,                //哈希函数
    ...
};
```

我们再进一步查看 dictSdsHash 函数的实现，可以发现它会调用 dictGenHashFunction 函数（在 dict.c 文件中），而 dictGenHashFunction 函数又会进一步调用 siphash 函数（在 siphash.c 文件中）来实际执行哈希计算。所以到这里，我们就可以知道全局哈希表使用的哈希函数是 **siphash**。

下面的代码展示了 dictSdsHash 函数及其调用的关系，你可以看下。

复制代码

```
uint64_t dictSdsHash(const void *key) {
    return dictGenHashFunction((unsigned char*)key, sdslen((char*)key));
}

uint64_t dictGenHashFunction(const void *key, int len) {
    return siphash(key,len,dict_hash_function_seed);
}
```

其实，Redis 源码中很多地方都使用到了哈希表，它们的类型有所不同，相应的它们使用的哈希函数也有区别。在 server.c 文件中你可以看到有很多哈希表类型的定义，这里面就包含了不同类型哈希表使用的哈希函数，你可以进一步阅读源码看看。以下代码也展示了一些哈希表类型的定义，你可以看下。

复制代码

```
dictType objectKeyPointerValueDictType = {
    dictEncObjHash,
    ...
}

dictType setDictType = {
    dictSdsHash,
    ...
}

dictType commandTableDictType = {
    dictSdsCaseHash,
    ...
}
```

第 4 讲

问题：SDS 判断是否使用嵌入式字符串的条件是 44 字节，你知道为什么是 44 字节吗？

这个问题，不少同学都是直接分析了 redisObject 和 SDS 的数据结构，作出了正确的解答。从留言中，也能看到同学们对 Redis 代码的熟悉程度是越来越高了。这里，我来总结下。

嵌入式字符串本身会把 redisObject 和 SDS 作为一个连续区域来分配内存，而就像 @曾轼麟同学在解答时提到的，我们在考虑内存分配问题时，需要了解**内存分配器的工作机制**。那么，对于 Redis 使用的 jemalloc 内存分配器来说，它为了减少内存碎片，并不会按照实际申请多少空间就分配多少空间。

其实，jemalloc 会根据申请的字节数 N，找一个比 N 大，但是最接近 N 的 2 的幂次数来作为实际的分配空间大小，这样一来，既可以减少内存碎片，也能避免频繁的分配。在使用 jemalloc 时，它的常见分配大小包括 8、16、32、64 等字节。

对于 redisObject 来说，它的结构体是定义在server.h文件中，如下所示：

复制代码

```
typedef struct redisObject {
    unsigned type:4;    // 4 bits
    unsigned encoding:    //4 bits
    unsigned lru:LRU_BITS;  //24 bits
    int refcount;  //4字节
    void *ptr;    //8字节
} robj;
```

从代码中可以看到，redisObject 本身占据了 16 个字节的空间大小。而嵌入式字符串对应的 SDS 结构体 sdshdr8，它的成员变量 len、alloc 和 flags 一共占据了 3 个字节。另外它包含的字符数组 buf 中，还会包括一个结束符“\0”，占用 1 个字符。所以，这些加起来一共是 4 个字节。

复制代码

```
struct __attribute__ ((__packed__)) sdshdr8 {
    uint8_t len;         // 1字节
    uint8_t alloc;     // 1字节
    unsigned char flags; // 1字节
    char buf[];    //字符数组末尾有一个结束符，占1个字节
};
```

对于嵌入式字符串来说，jemalloc 给它分配的最大大小是 64 个字节，而这其中，redisObject、sdshdr 结构体元数据和字符数组结束符，已经占了 20 个字节，所以这样算下来，嵌入式字符串剩余的空间大小，最大就是 44 字节了（64-20=44）。这也是 SDS 判断是否使用嵌入式字符串的条件是 44 字节的原因。

第 5 讲

问题：在使用跳表和哈希表相结合的双索引机制时，在获得高效范围查询和单点查询的同时，你能想到这种双索引机制有哪些不足之处吗？

其实，对于双索引机制来说，它的好处很明显，就是可以充分利用不同索引机制的访问特性，来提供高效的数据查找。但是，双索引机制的不足也比较明显，它要占用的空间比单索引要求的更多，这也是因为它需要维护两个索引结构，难以避免会占用较多的内存空间。

我看到有不少同学都提到了“以空间换时间”这一设计选择，我能感觉到大家已经开始注意透过设计方案，去思考和抓住设计的本质思路了，这一点非常棒！**因为很多优秀的系统设计，其实背后就是计算机系统中很朴素的设计思想。**如果你能有意识地积累这些设计思想，并基于这些思想去把握自己的系统设计核心出发点，那么，这可以让你对系统的设计和实现有一个更好的全局观，在你要做设计取舍时，也可以帮助你做决断。

就像这里的“以空间换时间”的设计思想，本身很朴素。而一旦你能抓住这个本质思想后，就可以根据自己系统对内存空间和访问时间哪一个要求更高，来决定是否采用双索引机制。

不过，这里我也想再提醒你**注意一个关键点**：对于双索引结构的更新来说，我们需要保证两个索引结构的一致性，不能出现一个索引结构更新了，而另一个索引没有相应的更新。比如，我们只更新了 Hash，而没有更新跳表。这样一来，就会导致程序能在哈希上找到数据，但是进行范围查询时，就没法在跳表上找到相应的数据了。

对于 Redis 来说，因为它的主线程是单线程，而且它的索引结构本身是不做持久化的，所以双索引结构的一致性保证问题在 Redis 中不明显。但是，一旦在多线程的系统中，有多个线程会并发操作访问双索引时，这个一致性保证就显得很重要了。

如果我们采用同步的方式更新两个索引结构，这通常会对两个索引结构做加锁操作，保证更新的原子性，但是这会阻塞并发访问的线程，造成整体访问性能下降。不过，如果我们采用异步的方式更新两个索引结构，这会减少对并发线程的阻塞，但是可能导致两个索引结构上的数据不一致，而出现访问出错的情况。所以，在多线程情况下对双索引的更新是要重点考虑的设计问题。

另外，在同学们的解答中，我还看到 @陌同学提到了一个观点，他把 skiplist + hash 实现的有序集合和 double linked list + hash 实现的 LRU 管理，进行了类比。其实，对 LRU 来说，它是使用链表结构来管理 LRU 链上的数据，从而实现 LRU 所要求的，数据根据访问时效性进行移动。而与此同时，使用的 Hash 可以帮助程序能在 O(1) 的时间复杂度内获取数据，从而加速数据的访问。

我觉得 @陌同学的这个关联类比非常好，这本身也的确是组合式多数据结构协作，完成系统功能的一个典型体现。

第 6 讲

问题：ziplist 会使用 zipTryEncoding 函数，计算插入元素所需的新增内存空间。那么，假设插入的一个元素是整数，你知道 ziplist 能支持的最大整数是多大吗？

ziplist 的 zipTryEncoding 函数会判断整数的长度，如果整数长度大于等于 32 位时，zipTryEncoding 函数就不将其作为整数计算长度了，而是作为字符串来计算长度了，所以最大整数是 2 的 32 次方。这部分代码如下所示：

复制代码

```
int zipTryEncoding(unsigned char *entry, unsigned int entrylen, long long *v,
  …
  //如果插入元素的长度entrylen大于等于32，直接返回0，表示将插入元素作为字符串处理
  if (entrylen >= 32 || entrylen == 0) return 0;
  …
}
```

小结

好了，今天这节课就到这里。在前 6 讲中，我主要是给你介绍了 Redis 的数据结构，要想掌握好这几讲的内容，一个关键点是，你要理解这些数据结构本身的组成和操作，这样你

在看代码时，才能结合着数据结构本身的设计来理解代码的设计和实现，从而获得更高的代码阅读效率。

非常感谢你对课后思考题的仔细思考和认真解答。在看留言的过程中，我从大家的答复中看到了更加全面或是更加深入的代码解读，我自己受益匪浅。接下来，我还会针对剩余的课后思考题，以及同学们的提问来做解答。也希望你将仍然存在的疑问提到留言区，我们来一起交流讨论。

让我们将学习进行到底！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 28 | Redis Cluster数据迁移会阻塞吗？

下一篇 29 | 如何正确实现循环缓冲区？

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。